High Performance Secure Database Access Technologies for HEP Grids

Phase I Final Documentation

DOE Grant No. DE-FG02-05ER84369

SBIR Phase I
Scientific/Technical Document

PIOCON TECHNOLOGIES

April 17, 2006

Matthew Vranicar
John Weicher

SBIR Rights Notice

These SBIR data are furnished with SBIR rights under U.S. Deparment of Energy Grant No. DE-FG02-05ER84369. For a period of 4 years after acceptance of all items to be delivered under this grant, the Government agrees to use these data for Government purposes only, and they shall not be disclosed outside the Government (including disclosure for procurement purposes) during such period without permission of the grantee, except that, subject to the foregoing use and disclosure prohibitions, such data may be disclosed for use by support contractors. After the aforesaid 4-year period, the Government has a royalty-free license to use, and to authorize others to use on its behalf, these data for Government purposes, but is relieved of all disclosure prohibitions and assumes no liability for unauthorized use of these data by third parties. This Notice shall be affixed to any reproductions of these data in whole or in part.

Summary

The Large Hadron Collider (LHC) at the CERN Laboratory will become the largest scientific instrument in the world when it starts operations in 2007. Large Scale Analysis Computer Systems (computational grids) are required to extract rare signals of new physics from petabytes of LHC detector data. In addition to file-based event data, LHC data processing applications require access to large amounts of data in relational databases: detector conditions, calibrations, etc. U.S. high energy physicists demand efficient performance of grid computing applications in LHC physics research where world-wide remote participation is vital to their success. To empower physicists with data-intensive analysis capabilities a whole hyperinfrastructure of distributed databases cross-cuts a multi-tier hierarchy of computational grids (Figure 1). The crosscutting allows separation of concerns across both the global environment of a federation of computational grids and the local environment of a physicist's computer used for analysis [1].

Very few efforts are on-going in the area of database and grid integration research. Most of these are outside of the U.S. and rely on traditional approaches to secure database access via an extraneous security layer separate from the database system core [2], preventing efficient data transfers. Our findings are shared by the Database Access and Integration Services Working Group of the Global Grid Forum [3]: "Research and development activities relating to the Grid have generally focused on applications where data is stored in files. However, in many scientific and commercial domains, database management systems have a central role in data storage, access, organization, authorization, etc, for numerous applications."

There is a clear opportunity for a technological breakthrough, requiring innovative steps to provide high-performance secure database access technologies for grid computing. We believe that an innovative database architecture where the secure authorization is pushed into the database engine will eliminate inefficient data transfer bottlenecks. Furthermore, traditionally separated database and security layers provide an extra vulnerability, leaving a weak clear-text password authorization as the only protection on the database core systems. Due to the legacy limitations of the systems' security models, the allowed passwords often can not even comply with the DOE password guideline requirements. We see an opportunity for the tight integration of the secure authorization layer with the database server engine resulting in both improved performance and improved security.

Phase I has focused on the development of a proof-of-concept prototype using Argonne National Laboratory's (ANL) *Argonne Tandem-Linac Accelerator System* (ATLAS) project as a test scenario. By developing a grid-security enabled version of the ATLAS project's current relation database solution, MySQL, PIOCON Technologies aims to offer a more efficient solution to secure database access.

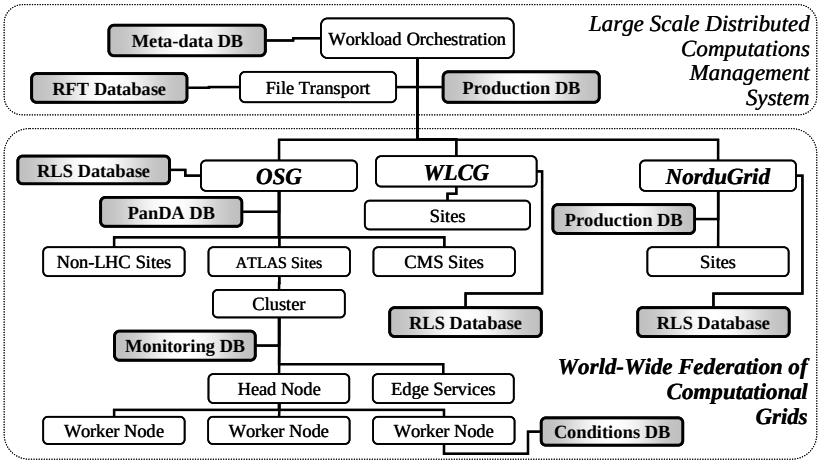

Figure 1. A hyperinfrastructure of distributed database services cross-cutting through a Large Scale Analysis Computer System of a federation of the computational grids.

I. Relevant Background

The ATLAS project relies on the resources of the Open Science Grid (OSG) for its complex data-analysis needs. In order for an entity to operate within the OSG, as well as be *authorized* to operate as a service on OSG, it must have a certified identity as a grid member. In the OSG, identities of both users and services are maintained and verified through the use of X.509 certificates, digitally signed using standard public-key cryptography techniques. To produce a grid-security aware application prototype, such functionality must be incorporated. Fortunately, through its inclusion of OpenSSL, an open-source implementation of Secure Sockets Layer (SSL), MySQL already has built in support for connections and communication established using these required technologies; this feature needs only to be enabled upon the building of the application.

Additional complications are present however, due to the very nature of grid operations. Rarely do typical grid activities involve only two parties communicating in one-to-one isolation. More commonly there is a need for the delegation of a user or service's rights or credentials to secondary services elsewhere within the grid, which may be called upon to handle a sub-component of a requested task. Such a delegation is handled through the use of a "proxy certificate". As its name implies, it is a temporary certificate derived from a permanent certificate, and digitally signed by the user or service on whose behalf it will be used. A proxy certificate may also have embedded within it additional information, such as additional security policy details, restrictions on where and how it may be used, etc.

The OSG infrastructure and its participating services are implemented using the Globus Toolkit, which is the de facto standard grid-framework API, and one which expectedly supports the use of proxy certificates. Unfortunately, the format of an OSG/Globus-compliant proxy certificate is different from that of the industry standardized X.509 proxy certificate, which the latest releases of OpenSSL and likewise MySQL already support. Therefore, in order to effectively grid-security enable MySQL, the primary goal of this project became the modification or extension of the MySQL application so that its certificate handling functionality could properly recognize and process Globus-style proxy certificates.

II. Component Details

MySQL

MySQL is a freely available open-source relational database application currently in use by the ATLAS project for the storage of calibrations and conditions data relevant to accelerator experiments. The data stored within this database is required when performing grid-based analysis. As with all grid operations, access to this service must be handled via a secure connection and secure authorization.

The MySQL application is structured such that if a connection between a client and server via a secure method is required, such a connection is managed through the "Vio", or "Virtual IO" module of the application. The Vio package serves to provide the rest of the application with a single interface to what may be a varied set of underlying connection infrastructures (sockets, named-pipes, etc.), depending on the platform on which MySQL is running. It is by routines found in this module through which an SSL connection is initiated and established. This process of secure connection negotiation is referred to as the "SSL Handshake".

OpenSSL

OpenSSL is an open-source toolkit which fully implements the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols for secure authorization and communication between two parties. The OpenSSL toolkit is a somewhat extensive package and is actually a collection of libraries, all of which are needed for an SSL framework to function. There are of course routines to handle SSL-specific transactions. Additionally, there is a library for the specific handling of X.509 certificates, which SSL relies upon, and which are defined by a rather large technical standard. Even further, there is a library which handles the encoding and decoding of arbitrary binary objects using Abstract Syntax Notation (ASN.1), yet another standard on which X.509 certificates and their subcomponents are structured.

A significant amount of time during the early portions of the project was spent on first mapping out the process and sequence of events that occur during the SSL handshake; we needed to determine how and where in the course of the handshake process our modifications would be required. This became a rather involved process because we were not initially experienced in SSL from a development standpoint, but also because of the size and complexity of the OpenSSL toolkit, as previously mentioned. Of course we would not need to fully understand in detail the inner workings of all these components to complete our project, and so were content to take much of them as is. However, teasing out what fit into this category, and then understanding as much as possible those aspects which could not simply be left "black-boxed", turned out to be unexpectedly time consuming.

Both during and after completion of examining the SSL handshake process, it was apparent that despite the logical symmetry between the actions of a client and server during this process, our focus would for the most part only need to be on the server side of the transaction. This is due to one simple observation: it is a client that is going to be presenting their identity via a proxy certificate, not a server. A server on a grid merely provides a service (in our case a database), and therefore never delegates its rights or credentials, and so never uses proxy certificates. Therefore, for our purposes only the server routines needed to be modified so that they could properly recognize and process a client's proxy certificate. A client needs only to be able to process a server's standard X.509 certificate.

The SSL Handshake

In a simplified description, the SSL handshake begins with a client connecting to a server and requesting the server's certificate. The client then attempts to verify the server's certificate (thereby its identity) by building a "certificate chain" for that server's certificate. After verification of the server certificate, which requires several steps itself, the server then has the opportunity to request the client's certificate, and do the same. Mutual authentication is not required by the SSL protocol,

but by default is enabled and performed by the implementation used by MySQL. This characteristic is fortunate, as mutual authentication is also required between hosts and services operating within the OSG. If mutual authentication is successful, identities of both parties have been verified, the connection is retained and communication can continue. If not, the connection is dropped (Figure 2).

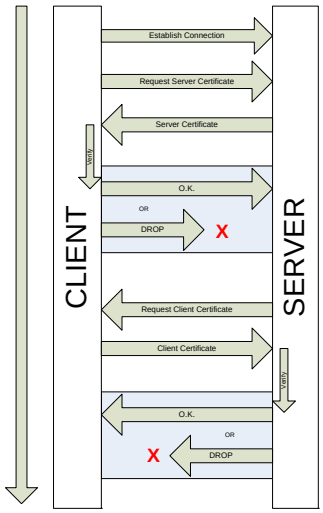

Figure 2: The SSL handshake process.

X.509 Certificates and Public Key Cryptography

To understand how certificate verification occurs, brief descriptions of what an X.509 certificate is and the role played by public-key cryptography are useful. Public-key cryptography is one of any number of cryptographic schemes that unlike earlier schemes do not rely on a single encryption/decryption key, but instead rely on a pair of keys. This pair of keys is two prime numbers that are generated using mathematical operations such that data encrypted with one number can only be decrypted with the other, and vice versa. Therefore, a party can publish one key (their "Public Key") with which any other party wanting to send a secure message can use to encrypt, and by keeping their second key (the "Private Key") secure, the recipient of such a message can guarantee that only they have the ability to decrypt and read the message (Figure 3).

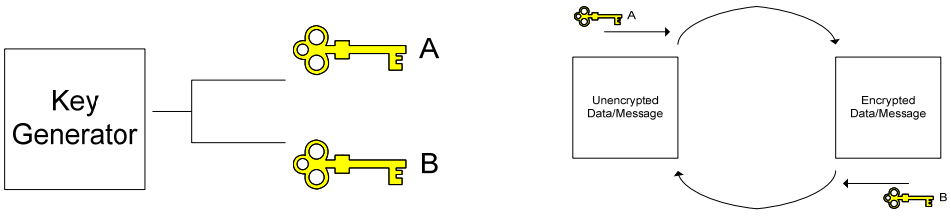

Figure 3: Public key pairs and usage.

The strength of public key cryptographic systems lies with the fact that the numbers generated and used as keys are extremely large. They are so large that a third party attempting to intercept a communication by taking a recipient's public key and attempting to derive the matching private key would find the numerical factorization necessary computationally infeasible.

An X.509 certificate is data that an entity can present which in effect says, "This is who I am claiming to be, and here is my public key." What makes this information reliable and trustable is that included with this identity information is the information for a third party, a Certificate Authority. This Authority is a higher level entity who issued the certificate, and the included

information contains their digital signature. This signature is a hash of the original certificate information that has been encrypted with the Authority's *private* key. This is effective because a recipient of a certificate can look at the signature of the authority who signed it, and if the authority is trusted by the recipient, the signature can be decrypted with their *public* key (which is publicly known). If the decrypted information is identical to original certificate information itself, the recipient is guaranteed that the trusted authority did in fact vouch for the presented certificate, and can in turn trust the presenter.

Verifying a Certificate: Building Certificate Trust Chains

At the core of the SSL certificate verification process is the requirement that an entity be able to build a "trust chain" for a certificate that it is presented. If this can't be accomplished, the certificate is rejected. All X.509 certificates must be signed by a Certificate Authority. Furthermore, even Certificate Authorities have their own certificates signed by even higher authorities. This system creates a hierarchy of identity validation. The central concept is that for every certificate an entity will accept, they must be able to trace the hierarchy of signatures upwards until one is encountered of a Certificate Authority that the entity already "knows" and trusts (Figure 4). This is accomplished by maintaining a local cache of certificates of higher authorities that are known to be trusted, which can be used for comparisons. Therefore, while mildly burdensome in actual execution, the SSL handshake is conceptually straightforward; two parties exchange certificates, both attempting to trace issuer signatures and certificates until one matches a certificate of a trusted authority contained in their local repository. If such matches are found, both parties have established trust.

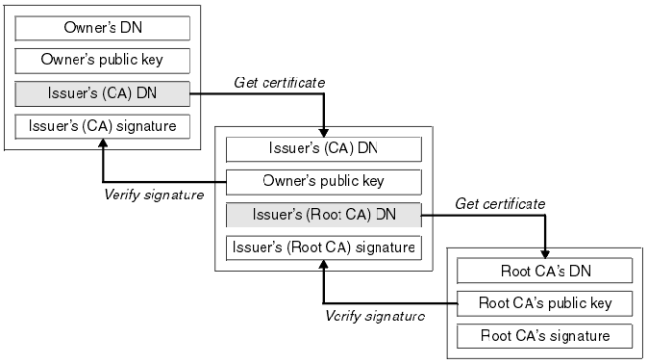

Figure 4: Certificate Trust Chain and process flow.

It is during this process of certificate chain building that errors are encountered when using Globus-style proxy certificates. The X.509 standard defines various properties and entries (referred to as "extensions") that must be defined for all valid X.509 certificates. There are also specific additional constraints which are defined for both proxy certificates and certificates authorities (in this case, any entity that takes on the responsibility of signing other certificates). While the overall structure and encoding process of Globus-style proxy certificates follow the guidelines laid out by the X.509 standard, some of these additional constraints and extensions differ. Because of these differences, Globus-style proxy certificates are rejected by the default OpenSSL verification procedures.

Aspect Oriented Programming

While not utilized until later iterations of the project, Aspect Oriented Programming (AOP) is a crucial technology and technique used in our prototype. AOP is a programming paradigm that allows for the modularization and encapsulation of cross-cutting concerns within an application or package. An example of such a concern would be the tasks of outputting debugging information, or event logging tasks within an application. An application likely has a consistent strategy for such tasks which "cuts across" modules

which themselves might be vastly different. An implementation of an AOP language tries to encapsulate these types of concerns through the introduction of a new Class-like construct called an “aspect”. An aspect can alter the behavior of base code (the non-aspect part of a program) by applying “advice” (additional behavior) over a quantification of “join points” (points in the structure or execution of a program), called a “pointcut” (a logical description of a set of join points) [4].

As MySQL is an application written in the C programming language, we chose to use the implementation of AOP, AspectC++, an open-source AOP compiler. With respect to the overall production pipeline, the AspectC++ compiler acts as an additional preprocessing automatic code-generation step, which takes un-compiled source code and inserts or “weaves” in advice code that we have defined (Figure 5). This weaved code can then be compiled using existing build methods and will have all our defined modifications incorporated.

In regards to Phase 1, our interest in AOP is not derived from the need to address truly cross-cutting concerns. However, it is extremely valuable from the perspective of being able to modularize our modifications away from the native MySQL and Globus source code. Whether preparing pre-built binaries of our finished product, or looking to supply a “patch” that a separate individual can easily apply to their own copy of the MySQL source code, AOP is a very efficient solution for introducing our modifications within requiring manual changes to any existing application source code.

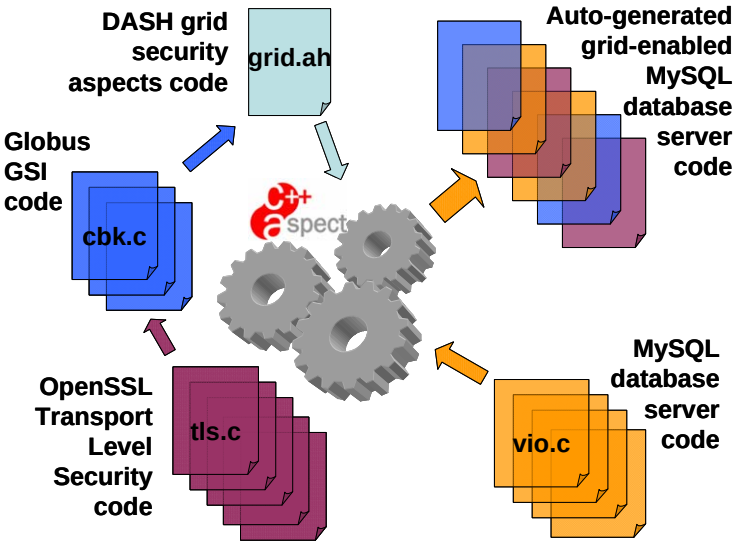

Figure 5: Aspect Oriented Programming compilation process.

III. Product Development Phases

Foundational Testing

We began the project by first ensuring that we could assemble the necessary components of the standard MySQL application, build them successfully and run the application on our development server. Before attempting any actual modifications to the application source-code we needed to make sure that there were no platform incompatibilities, or other preventable errors that might arise. As the end result of this project was a Grid-compatible application, this also meant performing our installation and configuration of the application so that it would include the support for SSL, as this is the mechanism through which OSG applications and services exchange and verify certificates between a client and a host.

In its conventionally available state, MySQL already includes support for SSL and X.509 certificates, through its inclusion of OpenSSL. Enabling support of SSL only required that we supply additional parameters to the installation and build utilities provided with the application. However, our aim was to produce a stable implementation of our product that was likely to be run on a variety of different Unix-based platforms. Because of this, we had to assume that end users might attempt to build our application source with various different versions of the OpenSSL libraries. Specifically, instead of using the OpenSSL libraries that are distributed with MySQL itself, some users may instead rely on the libraries that are distributed with the Globus Toolkit, the API that is used to build the majority of OSG applications, and the one that we ourselves would be incorporating. Therefore, it was prudent to perform multiple initial builds of MySQL, each using a different distribution of the OpenSSL libraries.

In total, we performed the following four initial builds of MySQL using different OpenSSL library distributions:

1) MySQL source with included OpenSSL libraries (v5.0.10-beta)
2) MySQL source with latest stable release OpenSSL libraries from http://www.openssl.org (v 0.9.8)
3) MySQL source with OpenSSL libraries included with the latest release of the Globus Toolkit 4 (GTK4.0.0)
4) MySQL source with OpenSSL libraries included with the current release of the OSG Service Suite installer (OSG v0.2.1, via Virtual Data Toolkit)

These four different builds gave us an acceptable sampling of the different distributions of the OpenSSL libraries that might be used by OSG sites and users who chose to build or use our product. As is implied by the above listing, it is during this stage that we secured the latest release of the Globus Toolkit (version 4). Initially we needed the OpenSSL libraries packaged with it; however we also needed its other components almost immediately thereafter, to begin our actual development.

As mentioned earlier in this document, we would eventually rely on Aspect Oriented Programming for the application of our modifications, while keeping our additions modularized. However, for our initial attempts, we decided not to concern ourselves with avoiding direct changes to the existing MySQL application code. Our foremost goal was to confirm, through any implementation, that we could successfully extend MySQL's existing functionality to support Globus proxy certificates. In our final product this would of course be unacceptable, as we do not want to have to convince the developers of MySQL to incorporate *our* changes into *their* application. Furthermore, this would be a very unacceptable approach from the standpoint of protecting our work and investment in the development of the product. However, for our initial attempts, concerns over implementation and packaging were overlooked.

Testing Native Support for SSL Authentication

When connections between a MySQL client and host must be handled securely, MySQL provides for the

ability to specify the "subject" that a connecting client's certificate must contain in order to connect using a specific account. The subject field of an X.509 certificate is an entry in the certificate that contains the unique, distinguished name of the individual to whom the certificate belongs (Figure 6).

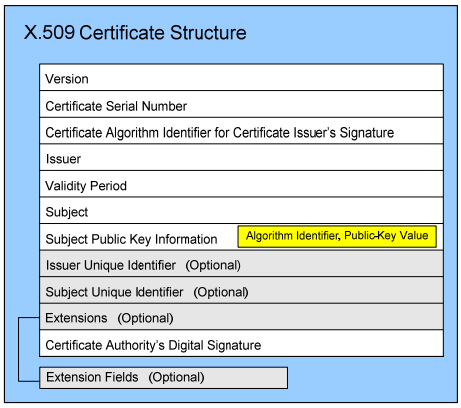

Figure 6: X.509 certificate structure.

One of our first tasks was to verify that this mechanism was working properly before proceeding with any modifications to MySQL. An account was created within MySQL for a user who could log in through the use of an X.509 certificate, in place of a standard username and password combination. The account specified the subject of one of our own personal certificates. This process involved creating a test database, and specifying the appropriate account access rights, through MySQL "GRANT" statements. While logged in via the root account, the following statements were issued to accomplish this:

```
> CREATE DATABASE gridTest;
> GRANT * ON gridTest.* TO john REQUIRE SUBJECT \
  "/DC=org/DC=doegrids/OU=People/CN=John C. Weicher 733602"
> FLUSH PRIVILEGES;
```

Once these tasks were completed, the process of then logging in via a standard X.509 certificate could be performed successfully by passing additional certificate-related command-line information to both the client and server applications regarding their respective certificates.

Client:

```
> client/mysql  \
     --ssl-cert=<path to user cert>/usercert.pem  \
     --ssl-key=<path to user key>/usercert.pem  \
     --ssl-capath=/etc/grid-security/certificates \
     --user=john  \
     --host=209.249.63.202
```

Server:

```
> sql/mysqld  \
     --basedir=<path to install dir>  \
     --datadir=<path to install dir>/data  \
     --ssl-capath=<path to CA dir>  \
     --ssl-cert=<path to host cert>/hostcert.pem  \
     --ssl-key=<path to host key>/hostkey.pem  \
     --skip-innodb  \
     -L <path to install dir>/sql/share/english/
```

The SSL handshake occurred without errors, and access was allowed via the requested MySQL account.

After verifying a successfully connection, we then created an additional account for a user to log in via a proxy certificate. We granted access rights in much the same fashion this time using the "issuer" field to restrict access:

```
> GRANT * ON gridTest.* TO johnproxy REQUIRE ISSUER \
"/DC=org/DC=doegrids/OU=People/CN=John C. Weicher 733602"
```

When starting the MySQL client, a proxy certificate file was used in place of the original certificate when specifying startup parameters. However, upon testing access for this account, proxy certificates were rejected and caused errors. This was expected and led us to our first attempt at modifying the application code to deal with the problem.

3.1 Using the Globus Callback Functions

Our first attempt at providing support for proxy certificates was to address an important component of SSL certificate verification, the use of callbacks. The OpenSSL implementation allows for the specification of "callback functions"; user-defined functions which are called at various points during the verification process of a certificate, which handle errors and through which additional steps of verification that might be required by a specific application can occur. Furthermore, because these callback operations occur both during and after the standard verification operations handled by OpenSSL, they can even serve to override or change verification decisions made by OpenSSL. As all OSG applications developed with the Globus Toolkit rely on SSL for the negotiation of secure connections, the Toolkit already contains its own callback functions that can be used in place of the native OpenSSL callbacks, to handle and if appropriate override, the proxy-related errors that occur during the handshake process. Our hopes were that by simply designating the use of the Globus callbacks and rebuilding the application source, our goal would be reached. This initial attempt involved modifying only a small number of actual lines of code.

3.1.1 Code Modifications

The following documented modifications were made to the MySQL Vio module code:

viosslfactories.c

```
+18   #include "globus_gsi_callback.h"

+363  SSL_CTX_set_cert_verify_callback(ptr->ssl_context,
      globus_gsi_callback_X509_verify_cert, NULL);

-370  SSL_CTX_set_verify(ptr->ssl_context, verify, vio_verify_callback);
+370  SSL_CTX_set_verify(ptr->ssl_context, verify, grid_verify_callback);
```

viossl.c

```
+25   #include "globus_gsi_callback.h"

+90   EVP_set_pw_prompt("Enter GRID pass phrase:");

-91   if (SSL_CTX_use_certificate_file(ctx,cert_file,SSL_FILETYPE_PEM) <= 0)
+91   if (SSL_CTX_use_certificate_chain_file(ctx,cert_file) <= 0)
+282  globus_gsi_callback_data_t callback_data;

+310  if (globus_gsi_callback_data_init(&callback_data) != GLOBUS_SUCCESS)
     {
       DBUG_PRINT("error", ("globus_gsi_callback_data_init failure"));
       DBUG_RETURN(1);
     }
     SSL_set_app_data((SSL*) vio->ssl_arg, &callback_data);
```

As seen above, a few other minor changes were also made in addition to the callback function substitutions. Specifically, the file containing the proxy certificate information to be used by a client contains data not only for the proxy certificate itself, but also for the standard certificate from which it is derived; it in effect contains a short certificate *chain*. Therefore, the appropriate routine must be used to load the entire chain of certificates, and not simply the proxy certificate. Furthermore, the Globus callbacks require some additional data structures that needed to be created and initialized, which are required by some internal functionality of the callbacks.

3.1.2 Results

Despite our hopes, this initial attempt did not yield the results that we were ultimately aiming for, though some progress was made. The application was rebuilt successfully after the inclusion of the Globus callback functions. When starting the new client and server, we supplied the client application with a proxy certificate file instead of a standard X.509 certificate. During the SSL handshake process, the proxy certificate was this time accepted and access into the application was granted. However, after examining our debugging output, it was apparent that the process had problems. While at first it appeared that the Globus callback functions were properly intercepting and overriding the error generated by the OpenSSL routines due to the use of an unsupported proxy certificate, allowing the process to continue, further examination of debugging information showed that in actuality, it proved to be the case that the entire trust chain for the client's certificate was not being built, and that in fact no certificates were being verified at all. Furthermore, after subsequent attempts at authenticating via a proxy certificate, it was also seen that regardless of correctness, there appeared to be no mechanism operating that checked for expired certificates.

There appeared to be several aspects of this attempt that we would need to examine more fully. At this point we decided that our next attempt would focus on verification of the correct loading of a certificate chain, enabling expiration-date checking for all certificates, as well as the implementation of additional debugging output, so that we could verify what in fact was occurring during the certificate chain construction. This would be a necessary step in confirming that proper verification was indeed proceeding.

3.2 Wrapping the Globus Callbacks – Extended Debugging Output, Expiration Date Check

To address the issues encountered during our first attempt, we decided not to modify any existing functions, but to instead create additional methods, which would "wrap" the Globus callback functions, and extend their functionality. These functions would pass execution through to the Globus callbacks, while also executing any additional operations, such as increased debugging output or expiration date verification. This was also a desirable approach because while we were still manually modifying small pieces of MySQL source code, something we would later want to move away from entirely, we would not be altering source code from any of the Globus routines.

There are two primary callback functions that are used by the OpenSSL process, and for which the Globus team created their own wrappers. These callback functions initiate and direct the actual certificate verification process while performing some specific operations themselves, as well as handle major errors; respectively, they are `globus_gsi_callback_X509_verify_cert()` and `globus_gsi_callback_handshake_callback()`. Our approach was to in turn wrap these functions to provide our own extended functionality.

Our first wrapper function `mysql_grid_handshake_callback()`, consists of operations to print the subject and issuer of the certificate for which the callback was executed, then pass execution on to the Globus function, and return its result. Because this callback is called at least once for every certificate processed (in the case of successful verification), and perhaps additional times (in the case of errors), this extended functionality would allow us to verify whether or not the entire certificate chain was being assembled and checked for a proxy certificate. This output information is missing by default from both the MySQL and Globus debugging information.

Our second wrapper function, `mysql_grid_X509_verify_cert()`, is used to wrap the functionality of the callback that is used at the beginning of the verification process for a certificate chain. Because of this, it is an ideal point in the process in which to implement our additional check for the expiration date of a clients certificate.

3.2.1 Code Modifications

The following is the content of our two Globus callback wrapper functions:

```
static int mysql_grid_handshake_callback(int ok, X509_STORE_CTX *ctx){
  char buf[256];
  X509 * curr_cert;  X509 * cert;
  int err, i;
  time_t ptime;
  X509_EXTENSION * ext;
  ASN1_OBJECT * obj;
  globus_result_t result;
  globus_gsi_cert_utils_cert_type_t cert_type;

#if GRID_DEBUG
  printf(">> Entering mysql_grid_handshake_callback()\n");
#endif
  curr_cert = X509_STORE_CTX_get_current_cert(ctx);
#if GRID_DEBUG
  X509_NAME_oneline(X509_get_subject_name(curr_cert), buf, sizeof(buf));
  printf("Cert Subject Name: %s\n", buf);
  X509_NAME_oneline(X509_get_issuer_name(curr_cert), buf, sizeof(buf));
  printf("Cert Issuer Name: %s\n", buf);
#endif
  ok = globus_gsi_callback_handshake_callback(ok, ctx);
#if GRID_DEBUG
  printf("<< Exiting mysql_grid_handshake_callback() [ok = %d (returnval), ctx = %p]\n",
ok, ctx);
#endif
  return(ok);
}
```

```
static int mysql_grid_X509_verify_cert(X509_STORE_CTX * ctx, void * args){
  int result, i;
  time_t ptime;
  X509 * curr_cert;

#if GRID_DEBUG
  printf(">> Entering mysql_grid_X509_verify_cert() [ctx = %p, args = %p]\n", ctx, args);
#endif
  result = globus_gsi_callback_X509_verify_cert(ctx, args);
  curr_cert = X509_STORE_CTX_get_current_cert(ctx);
  ptime = time(NULL);
  i = X509_cmp_time(X509_get_notBefore(curr_cert), &ptime);
  if(i == 0){
    ctx->error = X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD;
    ctx->current_cert = curr_cert;
    result = 0;
    goto exit;
  }
  if(i > 0){
    ctx->error = X509_V_ERR_CERT_NOT_YET_VALID;
    ctx->current_cert = curr_cert;
    result = 0;
    goto exit;
  }
  i = X509_cmp_time(X509_get_notAfter(curr_cert), &ptime);
  if(i == 0){
    ctx->error = X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD;
    ctx->current_cert = curr_cert;
    result = 0;
```

```
    goto exit;
  }
  if(i < 0){
    ctx->error = X509_V_ERR_CERT_HAS_EXPIRED;
    ctx->current_cert = curr_cert;
    result = 0;
    goto exit;
  }
  exit:
#if GRID_DEBUG
    printf("<< Exiting mysql_grid_X509_verify_cert() [ctx = %p, args = %p, result = %d
(returnval)]\n", ctx, args, result);
#endif
    return(result);
}
```

3.3 Custom Certificate Loading Routine

After creating and implementing our wrapper functions to provide expiration date checking and more robust debugging information, we performed another build. At this point, we were not expecting to actually produce a solution to our final goal, as our wrapper functions did not implement any new functionality that would *remedy* whatever errors were occurring. The expiration date checking was simply extended functionality beyond what was already not working correctly. However, the additional debugging output proved invaluable, as it immediately indicated that there were significant problems regarding the loading of the proxy certificate file and its chain. This in turn prompted us to again do some further examination with the GDB debugger. Upon doing so, we realized that the client's proxy certificate was not being correctly loaded at all. Because of this fact, it became mandatory that we no longer rely on the OpenSSL native routines for loading certificate information.

Our next iteration's focus would be to implement our own certificate loading routines using the Globus libraries for use by the client portion of the application. The Globus libraries contain routines necessary to correctly access and load the components of their proxy certificate format, and so could be used to properly parse the proxy certificate file and load the individual components of a certificate and its chain (if present).

3.3.1 Code Modifications

Our certificate loading routine (show here as part of an Aspect, covered later) is as follows:

```
SSL_CTX ** ctxRef;
    const char ** cert_fileRef;    const char ** key_fileRef;
    int * load_typeRef; int * resRef; int load_type; int i;
    SSL_CTX * ctx;
    const char * cert_file; const char * key_file;
    globus_gsi_cred_handle_attrs_t handle_attrs=NULL;
    globus_gsi_cred_handle_t handle=NULL;
    int result=0; int num=0; int numFlipped=0;
    STACK_OF(X509) *sktmp=NULL; STACK_OF(X509) *sktmpFlipped=NULL;
    X509 *x=NULL; X509 *xTest=NULL;
    ctxRef       = (SSL_CTX **)tjp->arg(0);
    cert_fileRef = (const char **)tjp->arg(1);
    key_fileRef  = (const char **)tjp->arg(2);
    load_typeRef = (int *)tjp->arg(3);
    resRef       = (int *)tjp->result();

    ctx = *ctxRef;
    cert_file = *cert_fileRef;
    key_file = *key_fileRef;
    load_type = *load_typeRef;

    EVP_set_pw_prompt("Enter GRID pass phrase:");

    if(cert_file==NULL){
```

```
    result = -1;
    goto exit;
  }
  if(key_file != NULL){
    printf("ASPECT: Loading certificate using standard routines.\n");
    result = vio_set_cert_stuff(ctx, cert_file, key_file);
    goto exit;
  }

  result = globus_gsi_cred_handle_init(&handle, NULL);
  if(result != GLOBUS_SUCCESS){
    result = -1;
    goto exit;
  }
  result = globus_gsi_cred_read_proxy(handle, cert_file);
  if(result != GLOBUS_SUCCESS){
    result = -1;
    goto exit;
  }
  result = SSL_CTX_use_certificate(ctx, handle->cert);
  if(!result){
    result = -1;
    goto exit;
  }
  sktmp = sk_X509_dup(handle->cert_chain);
  if(sktmp == NULL){
    result = -1;
    goto exit;
  }
  sktmpFlipped = sk_X509_new_null();
  if(sktmpFlipped == NULL){
    result = -1;
    goto exit;
  }

  num = sk_X509_num(sktmp);
  for(;;){
    if(num==0) break;
    x = sk_X509_value(sktmp, num-1);
    result = sk_X509_push(sktmpFlipped, x);
    if(!result){
      result = -1;
      goto exit;
    }
    num--;
  }

  if(ctx->extra_certs != NULL){
    sk_X509_pop_free(ctx->extra_certs, X509_free);
    ctx->extra_certs = NULL;
  }

  num=sk_X509_num(sktmpFlipped);
  for(;;){
    if(num==0) break;
    x = sk_X509_value(sktmpFlipped, num-1);
    if( !SSL_CTX_add_extra_chain_cert(ctx, x)){
      X509_free(x);
      result = -1;
      goto exit;
    }
    num--;
  }

  if(result = SSL_CTX_use_PrivateKey(ctx, handle->key) <= 0){
    result = -1;
    goto exit;
  }

  if (!SSL_CTX_check_private_key(ctx))
  {
```

```
      result = -1;
      goto exit;
    }

    exit:
      /* It's all just pointers, and all the certs need to
         be kept around... so no for now? */
      if(handle_attrs!=NULL){
        if(globus_gsi_cred_handle_attrs_destroy(handle_attrs) != GLOBUS_SUCCESS) ;
      }

      if(result == -1)
        printf("Error: Unable to load user certificate and key.\n");
      *resRef = result;
}
```

3.3.2 Results

The replacement of the native OpenSSL certificate loading routine with another that used Globus functionality proved to be the most important component of our modifications, and the one that ultimately secured the results we were looking for in our proof of concept. After rebuilding the application, the SSL handshake process occurred as we had hoped. Using the extended debugging information we confirmed that the MySQL client application could now correctly identify a proxy certificate file, and use our Globus-based loading procedures in place of the native OpenSSL procedures when necessary.

With the proxy certificate sent by the client now being properly loaded, the server no longer rejected it as being improperly formatted. The Globus callback wrapper functions intercepted any errors and allowed the process to continue if an error was due to OpenSSL's inability to recognize attributes of the proxy certificate, and handled the appropriate verification themselves. Additionally, we ran multiple test connections using various expired and un-expired certificates of both the proxy and standard variety, and confirmed that our expiration date checking mechanism was also functioning properly. Certificates with validity periods that had expired were properly rejected.

3.4 Aspects

The result of the previous iteration proved to be a working, albeit unrefined, implementation of our project goal. Our next step was to address any other design and implementation concerns, specifically the modularization of our code modifications away from the native MySQL source code. To do this we set out to translate our code into aspects, which through AOP translation operations could then be weaved into the application source at compile time, eliminating the need to manually alter any MySQL source code directly.

In practice, aspects are implemented as C++ style objects, and themselves can contain multiple "advice" definitions, which are similar to object methods in function but with a differing syntax. This syntax, interpretable by the aspect pre-compiler, identifies an advice block not as a standard object method (which aspects can in fact contain; they are like any other C++ object), but as a block of code to be inserted in the manner dictated by the advice declaration.

Advice definitions have the following general form:

advice { **call | execution** }(<*reg. expression*>)
[&& **within**(<*reg. expression*>)] :
{ **before | after | around** }()
{

. . . *Advice Body* . . .

}

As the definition above describes, an advice definition contains at least two components. The first component dictates whether the advice will be applied against a single-line function call or against the execution *internal* to a function call (after it has been called). The function to which the advice should be applied is described by a type of regular expression. Given the preceding context, the second component of an advice definition dictates whether the advice application should occur before, after, or all together in place of that context. An optional middle component allows for the further limiting of an advice's application, by defining another function within which the context must be located in order for the advice to be applied.

4.1.1 Technical Details

Our implementation utilized only one aspect object, which itself contained numerous advice definitions that were sufficient in applying all our desired modifications. As a precaution a handful of operations were handled in functions located external to, but referenced by the aspect object, for reasons of scope. Our components are listed below, with brief descriptions of their purpose.

4.1.1.1 External Functions

static int mysql_grid_handshake_callback(…)

- This function is used as a wrapper to the Globus handshake callback, which itself replaces the native OpenSSL callback. Our extension serves to supply additional debugging output about the certificate currently being processed, and then passes control to the Globus callback.

static int mysql_grid_X509_verify_cert(…)

- This function is used as a wrapper to the native OpenSSL certificate verification process callback function. Our extension serves to first allow verification to proceed as normal, and in the case of successful verification, enforce expiration date checking of the relevant certificate, which by default is disabled by OpenSSL.

static void grid_set_callback_data(…)

- This function serves to perform creation, initialization and verification of a data structure which is required by the Globus callback routines in order to verify proxy certificates, and which must be inserted into an existing SSL data structure.

4.1.1.2 Advice Definitions

advice call("% %SSL_CTX_set_verify(…)") &&
within("% %new_VioSSLAcceptorFD(…)") :
around()

- This advice block serves to take the place of an existing call to MySQL's SSL_CTX_set_verify() call. This advice sets various options for the SSL session that otherwise would not be set. It also configures the session to use our callback wrapper

functions in place of the native callbacks. This advice affects the behavior of the MySQL server application.

advice call("% %vio_set_cert_Stuff(…)") &&
within("% %new_VioSSLConnectorFD(…)") :
around()

- This advice block serves to take the place of an existing call to MySQL's vio_set_cert_stuff(), which is the default certificate-loading function. Our replacement leverages the Globus libraries to properly load a certificate file if it contains a proxy, otherwise control is passed onto the default MySQL call. This advice affects the behavior of the client application.

advice call("% %SSL_set_accept_state(…)") &&
within("% sslaccept(…)") :
around()

- This advice block serves to intercept an existing function call which sets configuration parameters for the server, in order to call our *grid_set_callback_data()* function after the configuration function has performed it's operations. This advice affects the behavior of the server application.

advice execution("% sslconnect(…)") :
before()

- This advice block simply stores a global reference to an internal MySQL data structure that is required later for hostname verification, but which becomes unavailable during the handshake process for reasons of scope. This advice affects the behavior of the client.

advice call("% SSL_do_handshake(…)") &&
within("% sslconnect(…)") :
around()

- This advice block serves to implement verification between the hostname presented in a server's certificate, and the server hostname as obtained through the TCP/IP connection data, to ensure they match. This operation guards against a server posing as another by using a stolen certificate. This feature is optional, but if included during a build, this advice affects the behavior of the client.

advice execution("% sslaccept(…)") :
around()

- This advice block serves to intercept the main SSL handshake function of the server, in order to surround it with the activation and deactivation calls required by the various Globus modules used for proxy certificate verification. This advice affects the behavior of the server.

References

[1] A. Vaniachine, D. Malon, M. Vranicar "Advanced Technologies for Distributed Database Services Hyperinfrastructure", preprint ANL-HEP-CP-04-107, submitted for publication in Proceedings of the Meeting of the Division of Particles and Fields of the American Physical Society (DPF2004) Riverside, CA, August 26-31, 2004

[2] P. Watson, "Databases and the Grid" in Grid Computing: Making The Global Infrastructure a reality, F. Berman, A.J.G. Hey, and G. Fox (eds.) Wiley, 2003

[3] http://www.gridforum.org/6_DATA/dais.htm

[4] O. Spinczyk, D. Lohmann, M. Urban, "Aspect C++: an AOP Extension for C++", in *Software Developer's Journal*, pages 68-76, 05/2005. http://www.aspectc.org/Publications.6.0.html